INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

PACO++: *A parallel object model for high performance distributed systems*

Christian Pérez Thierry Priol André Ribes

N°4960

Octobre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4960--FR+ENG

PACO++: A parallel object model for high performance distributed systems*

Christian Pérez Thierry Priol André Ribes

Thème 1 — Réseaux et systèmes
Projet PARIS

Rapport de recherche n°4960 — Octobre 2003 — 21 pages

Abstract: With the availability of high bandwidth wide area networks, it is nowadays feasible to couple several computing resources — supercomputers or PC clusters — together to obtain a high performance distributed system.

The question is to determine a suitable programming model that provides transparency, interoperability, reliability, scalability and performance. Since such distributed systems appear as a combination of distributed and parallel systems, it is very tempting to extend programming models that were associated to distributed or to parallel systems.

Another choice is to combine the two different worlds into a single coherent one. A parallelism oriented model appears more adequate to program parallel codes while a distributed oriented model is more suitable to handle inter-code communications.

We have designed the concept of parallel CORBA object to address this issue. It extends the CORBA specification with the notion of parallel object. A parallel CORBA object is a collection of CORBA object with a Single Program Multiple Data (SPMD) execution model.

This paper deals with PACO++, a portable implementation of the concept of parallel CORBA object. It examines how the different design issues have been tackled with. For example, scalability is achieved between two parallel CORBA objects by involving all members of both collections in the communication. Early experiments with a 1 Gbit/s WAN show scalable performance: an aggregated bandwidth of 874 Mbit/s has been obtained between two parallel CORBA objects made of twelve nodes each. Such a performance has been obtained while preserving the semantics of CORBA. For example, parallel CORBA objects are interoperable with standard CORBA objects via a proxy mechanism. Only the implementer of a parallel CORBA object and parallel-aware clients have to deal with parallel issues.

PACO++ is currently used as the high performance object platform for several multiphysics applications like hydrology and aircraft simulations.

Key-words: PACO++, CORBA, parallel object

* This work was supported by the Incentive Concerted Action “GRID” (ACI GRID) of the French Ministry of Research.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

(Résumé : tsvp)

PACO++ : Un modèle orienté objet pour les applications scientifiques distribuées haute performance[†]

Résumé : La simulation numérique prend de plus en plus d'importance dans la conception de nouveaux prototypes dans des industries telles que l'automobile ou l'aéronotique. Ces simulations multi-physiques sont réalisées grâce au couplage de plusieurs codes de calcul scientifique. Ces simulations sont devenues possibles grâce à l'émergence des environnements de calculs distribués haute performance. Ceux-ci, appelés grilles de calculs, sont composés de centres de calculs distants reliés par des réseaux longue distance haute performance. Afin de maîtriser la complexité de réalisation de ces applications, une approche orientée objet apparaît comme une solution prometteuse. Cependant, les modèles objets actuels ne proposent pas de support pour encapsuler efficacement des codes parallèles. Ce papier décrit PACO++, une extension parallèle portable pour CORBA, qui propose un modèle permettant d'encapsuler des codes parallèles dans des objets CORBA. Un prototype a été réalisé et a permis de valider le modèle proposé.

Mots-clé : PACO++, CORBA, objet parallèle

1 Introduction

Numerical simulation is playing an increasing role in the design process of many competitive industries such as automobile or aerospace. It is now possible to simulate various physical phenomena before making any real prototype. Such a situation was made possible by the availability of high performance computers that become cost-effective when based on standard technologies, like PC clusters.

However, the need to have more accurate simulation requires nowadays to couple several simulations together to study the interactions between several physics. Such multiphysics applications require huge computing power. Hopefully, with the availability of high bandwidth wide area networks, it is nowadays feasible to couple several computing resources — supercomputers or clusters of PC — together to obtain a high-performance distributed system. Such a computing infrastructure, which brings together distributed and parallel systems, is now being called a computational grid.

The question is to determine an adequate programming model to design multiphysics applications that will be run on such a computing infrastructure. Several efforts have been made to design code coupling tools or frameworks which do not put too much burden on the programmers. Most of these tools [1, 2, 3] are targeted to specific physics fields (climate, aerospace, etc), rely on parallel runtime (for example MPI- Message Passing Interface [4]) and provide their own abstractions (component or object models or more often an *ad-hoc* API) without adhering to existing standards. Thus, they have some limitations. To illustrate the situation, the execution of an application based on MpCCI [1] on a computational grid requires an MPI implementation able to spawn MPI applications on several grid resources whereas MPI has not been designed to handle such communications. Moreover, MPI is being internally used by a simulation code (for parallel processing) and between simulation codes (for distributed processing). Therefore, multiphysics applications based on such an approach are monolithic, mostly static and rather complex to maintain.

Most of the problems stem from the utilization of a parallel oriented middleware (MPI) for inter-code communication, whereas such interactions should be handle by a distributed oriented middleware. Therefore, we propose to handle them with a widely adopted distributed object model from the Object Management Group (CORBA) and to let it be used for the design of code coupling tools. As such our solution is not comparable to existing code coupling tools. It just provides the basic mechanisms to facilitate their design and their implementation.

In order to efficiently handle inter-code communications, we extend the CORBA (Common Object Request Broker Architecture) specification with the notion of parallel object. A parallel CORBA object is a collection of CORBA objects with a Single Program Multiple Data (SPMD) execution model. The objective is to enable an easy and efficient parallel code encapsulation into a CORBA object. This paper presents PACO++, a portable implementation of the concept of parallel CORBA object.

Section 2 introduces some motivating applications, briefly analyzes their requirements and motivates the use of a distributed object model. In Section 3, the concept of parallel object is defined. The presentation of previous works is followed by a discussion of their limitations. Section 4 presents PACO++ and examines how the different design issues have been tackled with. Some preliminary experimental results are reported in Section 5. Section 6 concludes the paper.

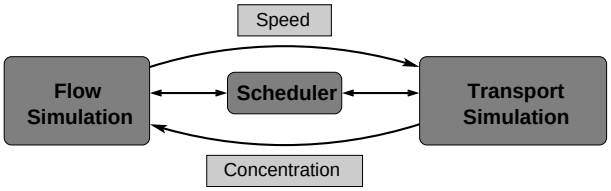

Figure 1: HydroGrid's code coupling schema.

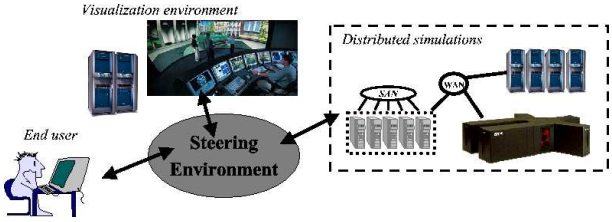

Figure 2: Overview of the EPSN project.

2 Motivating applications and analysis

2.1 Motivating applications

Though many kinds of applications may be concerned, we focus on multiphysics applications. Such applications consist in interconnecting several simulation codes more or less tightly. Throughout this paper, the term code denotes a set of functions in any programming language (FORTRAN, C, ...). Most of the time, simulation codes are parallel in order to handle the huge amount of computation or data required by a simulation. Let us introduce two representative examples.

Computational multiphysics application

An example of such application is given by the HydroGrid project [5] which aims at modeling and simulating fluid and solute transport in subsurface geological media. Figure 1 shows a simple multiphysics application with only two physics. Each physic is simulated by a dedicated code; the codes are developed by independent teams, in different languages: FORTRAN/MPI and OpenMP. A scheduler coordinates the control flows of the codes. As each code is only accessible by a public interface – ideally independent from the implementation language, each model can be discretized by different methods according to their physical properties, so as to conserve invariants, to respect maximum principles, etc. The transfers of the speed and concentration values, which may involve a huge amount of data, are in the critical path of the algorithm. Though latency-tolerant algorithms are

under investigation, a large amount of data will still need to be transferred within a bounded period of time.

Visualization and steering application

Another kind of application, driven by soft real time constraints, is given by the EPSN project [6]. Its goal is to analyze, to design and to develop a software environment for steering distributed numerical simulations from the visualization. As shown in Figure 2, it adds another level of constraints which stem from the human interaction loop: data have to be extracted from the application and sent to a visualization machine but also the user actions have to be sent back to the application. As a user can connect and disconnect to a running application from *a priori* any machine, a distributed-oriented middleware appears well-suited to handle such interactions.

2.2 Distributed execution environment

Multiphysics applications should be able to run in different environments. The general case is an heterogeneous distributed environment either to aggregate enough resources or to satisfy deployment constraints. The networks interconnecting the machines used by an application can be of various types, ranging from Internet to high bandwidth wide area networks. A noteworthy situation occurs when several codes are deployed into a same parallel machine or cluster: theses codes can utilize the local high performance network.

Another issue is the need for interoperability. For example, two organizations may temporally cooperate in a project without wanting to share their code. Hence, interoperability is required as each organization develops and deploys a part of the whole application.

In all these situations, the programming model should let the application run at the maximum efficiency allowed by the resources allocated to it.

2.3 Limits of the traditional code coupling approach

It is appealing to extend the approach used to program parallel machines to grids: merging the different codes into a single executable, generally based on MPI, directly or indirectly by a framework like MpCCI. Though, it is meaningful in some situations, this approach appears limited and too much restrictive for grids. Besides not supporting the two organization scenario presented hereinbefore, it obliterates the notion of code maintenance and evolution: it is not easy to integrate bug fixes or new versions of the original code into an existing multiphysics application.

Moreover, the application appears *closed*: it is not straightforward to connect the application to some external services like an ASP or to a steering and/or visualization environment like EPSN.

2.4 Distributed object oriented model

To handle the complexity of multiphysics applications, it appears important to use an adequate paradigm at each level. As it is obvious that computational code should be written according to a parallel paradigm – like message passing or a parallel language such as OpenMP, it does not seem

to be the case at the upper level. A distributed object oriented approach appears better suited to manage inter-code interactions.

Interoperability and dynamic object connection are required because they enable different organizations to set up a multiphysics application without sharing their codes. The localization transparency is also important as it eases the deployment.

Distributed object oriented standards like CORBA [7] appear very well suited as they are well established and widely available. However, with respect to our objective, communication performance and scalability need to be taken into account.

Communication performance issue

As the localization is determined at runtime, two CORBA objects can be interconnected through any kinds of networks, including high performance networks. Contrary to a widespread belief, we have shown [8] that CORBA communications can transparently and efficiently utilize all kinds of networks and in particular high performance networks: a latency of 20 μs and a bandwidth of 240 MB/s have been measured on a Myrinet-2000 network.

Scalability issue

The second issue, which is the topic of this paper, comes from the parallelism of the targeted codes. In our proposition, CORBA is in charge of handling communications between parallel codes. We assumed these codes have been deployed over different sets of machines. In the general case, the data to be sent (resp. to be received) are distributed over the machines of the sender (resp. receiver). A solution is to serialize the data and to use a standard CORBA invocation to transfer them. This solution is not satisfactory because the communication is a point-to-point communication. A mechanism being able to utilize all the sender and receiver machines is needed so that all machines can potentially participate to the communication: hence, high bandwidth and high performance networks will efficiently be exploited by aggregating the networking capabilities of all machines.

3 Distributed parallel objects

Before presenting the concept of (distributed) parallel objects, we briefly introduce parallelism.

Parallelism

It consists in several control flows that act together to realize an operation. In the general case, each control flow is located in a node and executes its own codes. While it is sometimes useful, this execution model, called Multiple Instruction Multiple Data (MIMD), is complex and does not appear adequate for massive parallelism.

Most parallel codes are written accordingly to a Single Program Multiple Data (SPMD) execution model: all control flows execute the same code but on different data. They generally communicate with global communication primitives such as broadcast, reduce, scatter, all-to-all, etc even though

point-to-point communications are possible. The most used parallel communication interface is the Message Passing Interface (MPI) standard [4].

Applications written with an SPMD paradigm usually deal with some global data such as matrices or graphs. These data are generally so large that they are scattered over the different processes accordingly to some data distribution functions. In the remainder of this paper, we will assume that processes are deployed over different nodes. So a N-node SPMD code means a code being executed by N processes located into N different nodes.

The question is to define an object oriented model that allows a parallel code to be embedded in a kind of object and that supports efficient and scalable communications between two such objects. That is to say, the model should define how a N-code SPMD code invokes a method of a parallel object provided by a M-node SPMD code. Moreover, as the arguments can be distributed either on the client side or on the server side, data redistribution issues shall be taken into considerations. A solution is brought with the concept of parallel object.

Definition

A (distributed) parallel object is a (distributed) object whose execution model is parallel. It is accessible externally through an object reference whose interpretation may be specific.

Definition

A (distributed) SPMD *object* is a (distributed) parallel object whose execution model is SPMD. A method invocation on such an object involves the coordinated invocation of the corresponding method in all nodes being part of the parallel object.

Consequences

A parallel object needs to be associated to several control flows. which are located into different nodes. This association can be static, for example defined at the object creation, or it can be dynamic, i.e it can be dynamically modified, for example to support adaptation. Moreover, a parallel object may require some communication capabilities to manage internal communications such as synchronizations or data redistribution issues. Last, if the parallel object model supports distributed arguments, it has to define how a parallel client declares parameters as distributed and how a parallel object declares the data distribution the arguments are expected to conform to.

3.1 Related work

There have been several works that have extended CORBA with the concept of parallel object.

PARDIS

The PARDIS CORBA-based environment [9, 10] is one of the first attempts to introduce parallel object in CORBA. It defined a new kind of object, called an SPMD object, which is an extension of

Listing 1: PARDIS example

```
interface diffusion {
  typedef dsequence<double,1024,
          (BLOCK,BLOCK)> diffusion_array;
  void diffusion (in long timestep,
          inout diffusion_array myarray);
};
```

a CORBA object. Data distribution issues are managed by a modification of the IDL that provides a generalization of the CORBA sequence called *distributed sequence.* PARDIS targets to program with SPMD objects similarly than with standard CORBA object. Therefore, a SPMD object interface looks like a standard interface as shown in Listing 1. Binding to a SPMD object is carried out through a specific method `spmd_bind` which is a collective form of the usual `bind` method. PARDIS allows to overlap execution of the client and the server codes thanks to its ability to perform non-blocking invocations. Such asynchronous invocation mechanism is based on the returning of *future* [11] for `out` arguments.

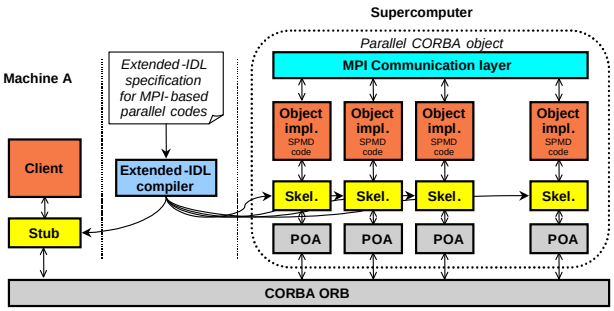

Figure 3: Parallel CORBA object.

Listing 2: Extended-IDL

```
interface [*] MatrixOperations {
  typedef double Matrix [ 1024 ] [ 1024 ] ;
  csum double det ( in dist [ BLOCK ]
          [ BLOCK ] Matrix A );
};
```

PACO

PACO [12, 13, 14], whose model is shown in Figure 3, is another early attempt for parallel programming in CORBA. To implement the concept of parallel CORBA object as defined hereinbefore,

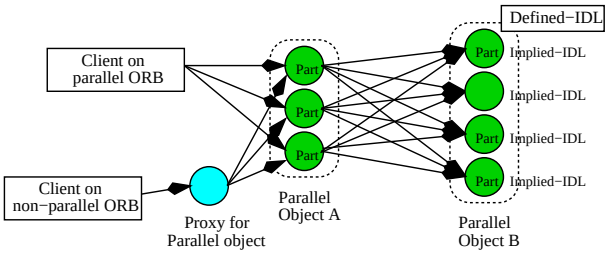

Figure 4: Data Parallel Object.

PACO extends the IDL language with new constructs. These constructs enable the specification of the number of CORBA objects being part of a parallel object and the data distributions of the operation parameters. PACO provides the data distributions defined in the High Performance Fortran language [15]. Stubs and skeletons which are generated by an *Extended*-IDL compiler use MPI operations to handle data redistributions. PACO allows collective operation to be performed with the return value of an operation as shown in Listing 2.

Data Parallel CORBA

More recently, the OMG has adopted a specification [16] that defines the architecture for data parallel programming in CORBA. Contrary to PARDIS and PACO, there is no definition of a parallel object at the IDL level but the specification enables to define the data and request distributions associated with a parallel object at runtime. Thus, it is a lower level mechanism that relies on a new POA (Portable Object Adapter) policy associated with a Parallel Part Adapter (PPA). This approach requires a specific ORB (parallel ORB) to manage parallel objects. As illustrated at Figure 4, a parallel object is seen as a collection of identical part objects. An IDL specification (called Defined-IDL) is associated with the parallel object and each part object is assigned with another IDL file (called Implied-IDL). This Implied-IDL is automatically derived from the Defined-IDL. Collective operation invocations are performed by the parallel-ORB. Invoking an operation of a parallel object from a standard ORB requires to use a proxy object which acts as a bridge between the different ORBs.

3.2 Discussion

The three previous approaches add support for parallel processing to CORBA at the expense at modifications to the CORBA standard. These extensions concern either the IDL language (PARDIS, PACO) or the ORB itself (Data Parallel CORBA). These approaches are unsatisfactory. Modifications of the IDL syntax require a new IDL compiler that is always dependent of a CORBA implementation. Therefore, such approach is not portable. Modification of the ORB is even worst because there are serious doubts that such extensions will be provided by numerous existing CORBA implementations.

The supported data distributions are hard-wired in the specification of the IDL for PARDIS and PACO. This choice leads first to the inability to dynamically change the data distribution, but more

important, it is almost impossible to efficiently write applications such as targeted by the EPSN project because specific data distributions are required. The OMG approach supports also a limited number of data redistributions. Moreover, it is a low level solution because it requires the implementers and the users of a parallel object to handle it.

4 PACO++: Portable Parallel CORBA Objects

PACO++ is the continuation of the PACO project as it shares the same parallel object definition and the same objectives. However, it supersedes PACO in several points. PACO++ targets to extend CORBA but not to modify the model because we aim at defining a *portable* extension to CORBA so that it can be added to any implementation of CORBA. This choice stems from the consideration that the parallelism of an object appears to be an implementation issue of the object. Thus, the OMG IDL is not required to be modified.

Like PACO, PACO++ is currently restricted to only support *Single Program Multiple Data* (SPMD) execution model. This choice stems mainly from the considerations that most parallel codes we target are indeed SPMD.

4.1 Outline

The PACO++ object model is defined in Section 4.2. It allows SPMD code to be embedded into a parallel object. The parallelism is supported thanks to a software layer, hereafter called the PACO++ layer. This code is inserted between the user code and the CORBA stub/skeleton code. This layer intercepts user's CORBA invocations to manage parallel calls issues. The layer's roles are described in Section 4.3.

A PACO++ object is interoperable with *standard* CORBA objects. For example, a standard CORBA client can invoke operations of a parallel object without noticing it is *parallel*. Similarly, a parallel code can invoke standard CORBA object operations. Section 4.4 describes the two new objects that enable such interoperability.

These two mechanisms, the PACO++ layer and the two proxies, allow the connection and the communication between two parallel objects to be managed. Section 4.5 describes the interactions between the proxies and the PACO++ layer to achieve the configuration of the client and of the server objects.

An important issue is the management of data redistributions. Ideally, any data distribution should be possible and should be dynamically exchangeable. Section 4.6 deals with the support of data distribution libraries as plug-in.

PACO++ integrates several invocation semantics as explained in Section 4.7 so as to control invocation synchronizations. The code generation schema of PACO++ is presented in Section 4.8. Finally, Section 4.9 presents an example that illustrates the user view.

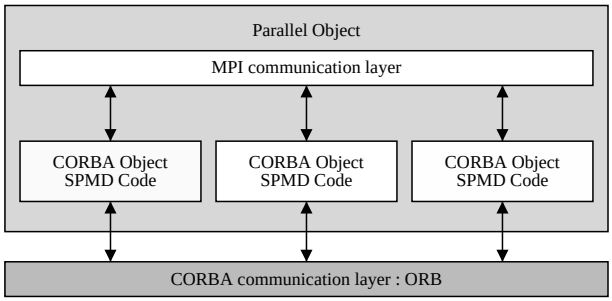

Figure 5: PACO++ object concept

4.2 A PACO++ parallel object

PACO++ keeps the definition of a parallel object from PACO. A PACO++ object is an SPMD CORBA object, i.e. a parallel object with an SPMD execution model. It is defined as a collection of identical CORBA objects.

Figure 5 shows the PACO++ view of a parallel object in the CORBA framework. The SPMD code is still able to use a communication library such as MPI for its intra-object communications but it uses CORBA to communicate with other objects. In order to avoid bottlenecks, all nodes of a parallel object can participate to inter-object communications. This model is in concordance with our constraints since the structure of parallel codes are not changed. Only some CORBA code is added to communicate with other objects.

Since a PACO++ object is a collection of objects, a representation of the whole object state in each node is needed to have a coherent behavior of the parallel object. To satisfy this constraint, a PACO++ object is associated with two kinds of context: a global context and an operation context. Each PACO++ object has a global context which represents the global characteristics of the object independently of parallel operations. This context contains information like the number of nodes, the communication library to use, etc. For each parallel operation, an operation context is created. With this context, the user can define the kind of distribution libraries (described in Section 4.6) to use, initialize these libraries, etc.

To sum up, a PACO++ object is a collection of CORBA objects that share contexts and that are able to invoke and to receive parallel CORBA invocations.

4.3 The PACO++ software layer

PACO++ defines a new software layer, called the PACO++ layer, to enable a management as transparent as possible of the parallelism.

A PACO++ layer is inserted between the client code and the CORBA stub code. In the same way, a PACO++ layer is inserted between the server code and the CORBA skeleton code. This is illustrated in Figure 6.

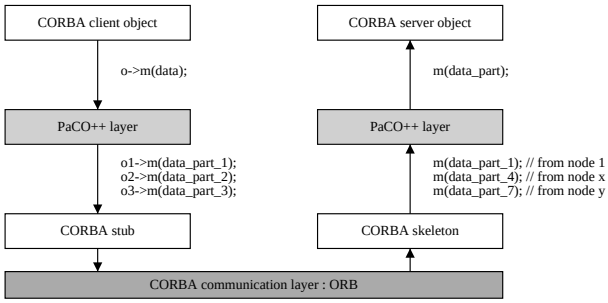

Figure 6: The user invokes an operation of a remote object. The PACO++ layer actually invokes the distributed version of this operation.

An invocation to a parallel operation of a parallel object is intercepted by the PACO++ layer at the client side. The distributed data are handled by the distribution libraries. Then, the layer invokes the corresponding operation of the PACO++ layer of the objects being part of the parallel object. The role of this operation is to wait the receipt of all data before locally invoking the object implementation code. The PACO++ layer uses the context information to perform parallel invocations and receptions.

The PACO++ layer needs parallel communication capabilities like barriers, broadcasts and reductions. Since the execution environment is not known, the fabric design pattern [17] is used to abstract the environment. Hence, abstract interfaces are used to handle communication and thread libraries. They define a portability API for PACO++. It is then possible to associate an actual communication library like MPI or PVM to the PACO++ layer either at code generation time or at run time.

4.4 A portable CORBA extension

One of the PACO++ objective is to be portable through different CORBA implementations. That is another reason not to modify the CORBA ORB nor the IDL. We choose to describe the parallelism of a parallel object in an auxiliary XML file. This file contains a description of the parallel operations of the object, the arguments that are distributed, and optionally the data distribution of the arguments. An example is given in Section 4.9. The XML file is used for generating the PACO++ layer (see Section 4.8 for more details).

Another objective is to let a parallel object be interoperable with standard CORBA objects. A parallel object must be able to invoke an operation on a standard CORBA object and it must be able to be invoked from a standard CORBA client. To support these two kinds of interoperability, PACO++ defines two new objects: the *Interface Manager* and the *Output Manager* as shown in Figure 7. The Interface Manager acts as a proxy that provides a standard CORBA interface from a parallel object. The reference of a PACO++ object is a reference to its Interface Manager. Hence, a PACO++ object appears like a standard object: its reference can be stored into a naming service.

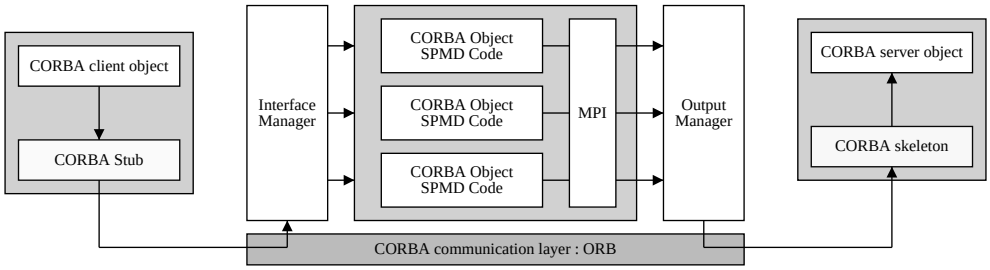

Figure 7: PACO++ proxies: the Input Manager and the Output Manager.

The Output Manager allows a parallel code (client or server) to collectively invoke an operation on a standard CORBA object. It intercepts the invocations of all instances of the parallel code to have only one invocation to the standard CORBA object.

4.5 The parallel object connection

This section deals with the creation and the configuration of the PACO++ layer. A PACO++ aware client is defined as a sequential or parallel code that wants to use the PACO++ layer.

In CORBA, a client generally gets a reference that needs to be downcasted into the right type with a *narrow* operation. In PACO++, the *narrow* keeps the same semantic: it creates a CORBA stub. To enable the PACO++ layer, a PACO++ aware client must use the *paco_narrow* operation. From the client point of view, this operation has the same behavior than the *narrow* operation: it returns a reference to a local proxy of the CORBA object. However, *paco_narrow* needs to identify the nature of the referenced object to create the right stubs. If the object is a standard CORBA object, *paco_narrow* just creates an output manager. If the object is a PACO++ object, it creates a PACO++ layer. As the reference is the one of the input manager, information related to the parallel object can be collected to correctly initialize the client PACO++ layer. Figure 9 illustrates the connection and the communication phases for a parallel client and a parallel object. Hence, a PACO++ aware client (sequential or parallel) can make use of it to invoke an operation on standard or parallel objects.

Table 8 sums up the different objects involved in all scenarios of client-server connections.

	Standard CORBA object		PACO++ object	
	client-side	server-side	client-side	server-side
Standard CORBA client	-	-	-	IM
Sequential PACO++ aware client	OM	-	L	L
Parallel PACO++ aware client	OM	-	L	L

Legend: **IM**: Interface Manager, **OM**: Output Manager, **L**: PACO++ Layer

Figure 8: Objects used in the different scenarios of the client and the server connections.

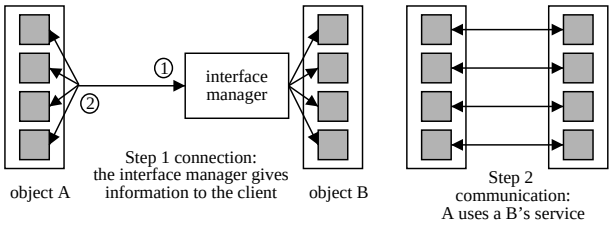

Figure 9: An example of connection and communication between two parallel objects A and B.

4.6 The data distribution management

One reason of the failure of the High Performance Fortran language [15] was its fixed number of data distributions. Another reason was related to its static data distribution type system. A goal of PACO++ is to be a research platform to study the management of data distribution libraries as a plug-in. Data redistribution libraries should be easily integrated into PACO++.

For performance and adaptability issues, PACO++ allows the data redistribution library to be dynamically selected and changed. To simplify its utilization, it is possible to statically specify it into the XML file.

This flexibility is achieved with the help of a fabric design pattern. All distributed data are instances of an abstract type that allows basic management operations. At the initialization phase, the client and server deployment code create the concrete type which currently has to be in concordance with the type defined in the IDL. A data distribution library provides two kinds of API. The first one is the API used by the application to describe its distributed data. This API is not seen by PACO++. The second API is dedicated to PACO++. Its role is to allow the PACO++ layer to extract/insert data and to determine the data to be sent/received.

PACO++ allows to express where the data redistribution is actually performed: either on the client side, or on the server side or during the communication between the client and the server. The decision depends on several constraints like feasibility (mainly memory requirements) and efficiency (client network performance versus server network performance).

4.7 Invocation semantics

It is important to support synchronous and asynchronous semantic invocations as it is a key feature to support different kind of applications.

The classical synchronous invocation is the default assumed by PACO++ as required by CORBA. However, some applications do not want to wait for the computation termination or want to deferred the result retrieval from the operation invocation. Hence, PACO++ supports asynchronous parallel invocations which is a kind of parallel extension to the CORBA Messaging specification [7].

PACO++ allows to choose invocation synchronization at the client and server sides. The client can choose whether a synchronization is required after an operation invocation. Without synchronization, an invocation terminates in a parallel client without taking into account the state of other

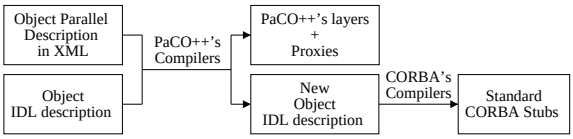

Figure 10: The PACO++ code generation schema.

Listing 3: An object IDL definition

```
typedef sequence<long> Vector;
interface myInterface {
  long compute(in Vector v);
};
```

nodes. Similarly, the objects being part of a parallel object can be synchronized on the beginning or the end of an operation invocation. This mechanism is similar to the one defines in the OMG data parallel specification [16].

4.8 The code generation schema

This section deals with the code generation schema used by PACO++ to generate the PACO++ layer and the proxies. Figure 10 displays a general view.

As most CORBA applications, an object designer defines the object interface in IDL. Then, as seen in Section 4.4, the designer describes the object parallelism in an XML file. A PACO++ specific compiler uses these two files to generate the proxies (the Interface and Output Managers) and the PACO++ layer for the client and the server. A PACO++ internal IDL description is also generated. This IDL contains the CORBA interface that is implemented by the PACO++ layer on the server side. It is used by the client side PACO++ layer to actually invokes a CORBA operation on the server. The standard CORBA stubs and skeletons are generated from this last IDL file for PACO++ aware clients and servers.

4.9 Example

This section gives an example of the definition and the use of a parallel object.

Listing 3 displays the IDL interface of a (parallel) object which provide an operation that computes the average of a vector. The object implementer needs to write an XML file that describes the parallelism of the object: the operation *compute* is declared parallel and its first argument is statically declared block distributed. This file, shown in Listing 4, is not presented in XML for the sake of clarity. An important remark is that this file is not directly seen by the client.

Listing 4: Parallelism description of the parallel object

Listing 5: A client initializes and uses a parallel operation

```
// Get a reference to the interface myInterface
PaCO_myInterface * avg =
        PaCO_myInterface::paco_narrow(obj);
// get and configure context
PaCO_context * global_cxt = avg->getGlobalContext();
// Configuration of the global context
...
// Configuration of the communication library
paco_fabric_com * fc = new paco_fabric_com();
fc->paco_register("mpi", new paco_mpi_fabric());
global_cxt->setComFab(fc);
MPI_Comm group = MPI_COMM_WORLD;
global_cxt->setLibCom("mpi",&group);
...
PaCO_context * compute_cxt =
        avg->getOperationContext("compute");
// Configuration of the operation context
compute_cxt->initArg(data_description,
        client_topology, 0);
...
// "Standard" Corba call
Vector MyMatrixPart;
...
avg->compute(MyMatrixPart);
```

```
Interface : myInterface
Operation : compute
Argument1: distributed
Result    : noReduction
```

A standard client will normally use the *myInterface* interface. However, a parallel client has to configure its contexts through to a client-side API. In Listing 5, the parallel client gets a reference of the interface *myInterface*. It uses the *paco_narrow* operation to create the PACO++ layer. Then, it configures the global and the operation contexts. The client can then invoke the *compute* operation on a parallel object.

5 Experiments

This section evaluates the performance of PACO++ when a parallel client invokes a parallel operation on a parallel object. All CORBA objects belonging to the parallel CORBA object are located on different machines. The tests are performed in two different deployment configurations. The first configuration is in an homogeneous cluster while the second configuration involves two clusters in-

terconnected by a high bandwidth wide-area network. Before presenting the experimental protocol and reporting some performance measurements, we need to introduce PadicoTM.

5.1 PadicoTM

PACO++ requires several middleware systems at the same time, typically CORBA and MPI. They should be able to efficiently share the resources (network, processors, etc) without conflicts and without competing with each other. These issues becomes very important if both middleware system want to use the same network interface, like for example TCP/IP. *A priori*, nothing guarantees that a CORBA implementation can cohabit with a MPI implementation. There are several cases that leads to application crashes as explained in [8].

In order to be sure to have a correct and efficient cohabitation of several middleware systems, we have designed PadicoTM [8]. PadicoTM is an open integration framework for communication middleware and runtimes. It allows several middleware systems (such as CORBA, MPI, SOAP, etc) to be used at the same time. It provides an efficient and transparent access to all available networks with the appropriate method.

PadicoTM enables us to simultaneously utilize CORBA and MPI and to associate both the CORBA and the MPI communications to the chosen network.

5.2 PACO++ prototype

The PACO++ compiler prototype is implemented in two languages : Java and Python. It generates the PACO++ layer in C++. This prototype implements most of the features described in this paper. We have tested the code generated with two ORBs : MICO and omniORB. Since omniORB is an efficient CORBA implementation, we used it for the experiments. Currently, PACO++ supports three thread libraries : POSIX threads, OmniORB Threads and Marcel Threads. Also, PACO++ supports two communication libraries : MPI and a PadicoTM specific one.

5.3 Experimental protocol

A first parallel object invokes an operation on a second parallel object with a vector of bytes as an argument. The invoked operation contains no code. Both parallel objects are deployed on the same number of nodes. The measures have been made in the first parallel object. After a MPI barrier to synchronize all nodes, the start time is measured. Client object performs 1000 calls to the server. Then, the end time is taken. The MPI communications always use a Myrinet network.

5.3.1 System Area Network Experiments

The test platform consists of dual-Pentium III 1 GHz with 512 MB of RAM, Linux 2.2, that are interconnected with a Myrinet-2000 network. The compiler is gcc 2.95.2. OmniORB3.0.2 [18] is used as the CORBA implementation.

A latency of 33 μs has been measured between a one-node client and a one-node server. The latency raises to 105 μs in a height-node client to height-node server configuration. The latency

Object node number	Aggregated bandwidth	Latency	
	VTHD (Mo/s)	SAN (μs)	VTHD (ms)
1	10.36	33	15
2	20.8	59	15
4	41.4	82	16
8	82.8	105	17
10	103.6	-	23
12	109.2	-	23

SAN: System Area Network
- : The local network has sixteen machines

Table 1: Aggregated bandwidth and latency between two parallel objects with identical number of nodes.

variation mostly reflects the variation of the time taken by the MPI barrier operation in function of the number of nodes. As the omniORB latency of top of PadicoTM is around 20 μs, it leads to an overhead of 13 μs for PACO++.

An aggregated bandwidth of 1.5 GB/s has been obtained in the 8-to-8 case, which represents a point-to-point bandwidth of 187 MB/s. It has been obtained without effective data redistribution. The bandwidth with effective data redistributions heavily depend on the data redistribution library, which is out of scope of this paper.

5.3.2 Wide Area Network Experiments

We have access to the VTHD network [19]. It is an experimental network of 2.5 Gb/s that in particular interconnects two INRIA laboratories, which are about one thousand kilometers apart. However, the sites were interconnected to VTHD with 1 Gbit/s switches.

A latency of 15 ms has been measured between a one-node client and a one-node server. It is the same latency than plain ombiORB between these two sites. The latency raises to 23 ms in a twelve-node client to twelve-node server configuration. This number seems too high to only depend on the MPI barrier. More experiments are needed to understand it.

In an experiment with a twelve-node client and a twelve-node server equipped with Fast-Ethernet card, we measure an aggregated bandwidth of 874 Mb/s (109.2 MB/s), which represents an average point-to-point bandwidth of 73 Mb/s (9.1 MB/s). The 1 Gb/s switch limits us to this bandwidth. PACO++ parallel objects prove to efficiently aggregate bandwidth.

6 Conclusion

Distributed systems enable new kinds of application thanks to the fast growth of high bandwidth wide area networks. A particular class of application is represented by multiphysics applications

that involve parallel codes and that require to transfer huge amount of data within a bounded period of time.

This paper focuses on a programming model which combines both distributed and parallel systems. The model is based on distributed parallel objects.

PACO++ applies this model to CORBA. It defines a parallel CORBA object as a collection of CORBA objects with an SPMD execution model. PACO++ is a portable and interoperable implementation build on top of standard CORBA implementations such as omniORB or MICO. It uses an XML auxiliary file to store information related to parallel issues. Moreover, data redistribution are not hard-wired in the model. PACO++ defines an API to interact with data redistribution libraries so that it is possible to add new ones.

More works involve a support of exceptions raised by parallel objects and a validation of the data redistribution API. Another work concerns the limitation of the consumed bandwidth. The model needs to respect the bandwidth allocated by some network quality of service or it should at least not saturate the backbone. A solution under development consists in integrating a communication scheduling algorithm [20]. A PACO++ prototype has been developed that implements most of the features described in this paper. PACO++ is used in several projects such as the HydroGrid and EPSN.

We have started to study the extension of the CORBA component model to support parallelism [21]. Our prototype GridCCM will be based on PACO++ as CORBA components are mapped at run time to CORBA objects.

References

[1] R. Ahrem, M.G Hackenberg, U. Karabek, P. Post, R. Redler, and J. Roggenbuck, "Specification of mpcci," Tech. Rep., GMD-SCAI, 2001.

[2] Sophie Valcke, Laurent Terray, and Andrea Piacentini, "Oasis user's guide and reference manual," Tech. Rep., CERFACS, Toulouse, France, 2000.

[3] "Ccsm user's guide," Tech. Rep., UCAR, 2002.

[4] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI : The complete reference*, MIT Press, 1995.

[5] "The HydroGrid Project," `http://www-rocq.inria.fr/~kern/HydroGrid/`.

[6] "The EPSN Project," `http://www.labri.fr/Recherche/PARADIS/epsn/index_eng.html`.

[7] OMG, "The Common Object Request Broker: Architecture and Specification (Revision 3.0.2)," Document – formal/2002-12-06, Dec. 2002.

[8] A. Denis, C. Pérez, and T. Priol, "PadicoTM: An open integration framework for communication middleware and runtimes," *Future Generation Computer Systems*, vol. 19, pp. 575–585, 2003.

[9] K. Keahey and D. Gannon, "PARDIS: A Parallel Approach to CORBA," in *Supercomputing'97*. November 1997, ACM/IEEE.

[10] K. Keahey and D. Gannon, "Developing and Evaluating Abstractions for Distributed Supercomputing," *Cluster Computing*, vol. 1, no. 1, pp. 69–79, May 1998.

[11] W.G. O'Farrell, F.Ch. Eigler, S.D. Pullara, and G.V. Wilson, *Parallel Programming Using C++*, chapter ABC++, MIT Press, 1996.

[12] T. Priol and C. René, "COBRA: A CORBA-compliant Programming Environment for High-Performance Computing," in *Euro-Par'1998: Parallel Processing*, Southampton, UK, Sept. 1998, vol. 1470 of *Lect. Notes in Comp. Science*, pp. 1114–1122, Springer-Verlag.

[13] C. René and T. Priol, "MPI code encapsulating using parallel CORBA object," in *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, Californie, USA, Aug. 1999, pp. 3–10, IEEE.

[14] Kamachi T., Priol T., and René C., "Data distribution for parallel CORBA objects," in *Euro-Par'2000: Parallel Processing*, Munchen, Germany, Aug. 2000, vol. 1900 of *Lect. Notes in Comp. Science*, pp. 1239–1250, Springer-Verlag.

[15] High Performance Fortran Forum, *High Performance Fortran Language Specification*, Rice University, Houston, Texas, Oct. 1996, Version 2.0.

[16] Object Management Group, "Data parallel CORBA," Nov. 2001, ptc/01-11-09.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison Wesley, 1995.

[18] "omniORB – Free High Performance ORB ," `http://omniorb.sourceforge.net`.

[19] "The VTHD project," http://www.vthd.org.

[20] J. Cohen, E. Jeannot, and N. Padoy, "Parallel data redistribution over a backbone," Tech. Rep. RR-4725, INRIA, feb 2003.

[21] C. Pérez, T. Priol, and A. Ribes, "A parallel corba component model for numerical code coupling," in *Proc. of the 3rd International Workshop on Grid Computing*, M. Parashar, Ed., Baltimore, MA, USA, Nov. 2002, number 2536 in LNCS, pp. 88–99, Springer-Verlag.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399